Accelerating Unstructured Mesh Computations using FPGAs

Kyrylo Tkachov

Supervisor: Professor Paul Kelly

May 24, 2012

Abstract

In this report we present a methodology for accelerating computations performed on unstructured meshes in the course of a finite volume approach. We use Field Programmable Gate Arrays, or FPGAs, to create an application-specific datapath that will be used as a co-processor to perform the bulk of the floating point operations required by the application. In particular, we focus on dealing with irregular memory access patterns that are a consequence of using an unstructured mesh in such a way as to facilitate a streaming model of computation. We describe the partitioning of the mesh and the techniques used to exchange information between neighbouring partitions, using so-called halos. We provide an implementation of a concrete 2D finite volume application and consider the extension to 3D and more complex computations. We evaluate our results by comparing the speedup achieved with analogous GPGPU and multi-core processor implementations.

Acknowledgements

I would like to thank Professor Paul Kelly for giving me so much of his time and ideas and making me aware of scope of the project and the intricacies involved. I would also like to thank Dr. Carlo Bertolli for providing practical advice and explaining the labyrinth that is heterogenous computing. Special thanks go to the team at Maxeler Technologies for helping me out with the details of FPGA-based acceleration and providing support for their excellent toolchain I extend my gratitude to Dr. Tony Field, my personal tutor, who supported me throughout my years at Imperial College and guided so much of my academic development, as well as being the second supervisor on this project.

I would like to thank my mother and grandfather for supporting me through university, both materially and psychologically. Last but not least, I would like to thank my coursemates and friends all over the world, with whom I've had many thought-provoking discussions on every subject imaginable and who always kept me motivated, even when I doubted myself.

Contents

Chapter 1

Introduction

This project presents a methodology for accelerating computations performed on unstructured meshes in the context of Computational Fluid Dynamics (CFD). We use Field Programmable Gate Arrays, or FPGAs, to construct a high-throughput streaming pipeline which is kept filled thanks to an appropriate data layout and partitioning scheme for the mesh. We explore the rearrangement and grouping schemes used to achieve locality of the data points. A formal performance model is constructed to predict the performance characteristics of our architecture and hence justify the design choices made. Appropriate evaluation tests are performed to evaluate the results on a sample CFD application, achieving speedup comparable with state of the art GPGPU and multi-processor solutions. In this section we present a general overview of the problem domain, the hardware platform and the contributions of this project.

1.1 The domain

Computational Fluid Dynamics, or CFD, is a branch of physics focused on numerical algorithms that simulate the movement of fluids and gases and their interactions with surfaces. These simulations are widely used by engineers to design structures and equipment that interact with fluid substances, for example airplane wings and turbines, water and oil pipelines etc.

The required calculations are usually expressed as systems of partial differential equations, the Navier-Stokes equations or the Euler equations, which are discretized using any of a number of techniques. The technique used by our sample application, Airfoil, is the finite volume method that

calculates values at discrete places in a mesh and relies on the observation that the fluxes entering a volume are equal to the fluxes leaving it. This project is not concerned with the exact mathematical formulation of these techniques, but they provide a feel for the origins of the problem domain.

1.2 The Airfoil program

The sample program we examine is called Airfoil, a 2D unstructured mesh finite volume simulation of fluid motion around an airplane wing (which has the shape of an airfoil). Airfoil was written as a representative of the class of programs that are tackled by OP2, a framework partially developed and maintained by the Software Performance Optimisation group at Imperial College to abstract the acceleration of unstructured mesh computations on a wide variety of hardware backends.

Airfoil defines an unstructured mesh through sets of nodes, edges and cells and associating them through mappings. Airfoil is written in the C language and these sets are represented at the lowest level as C-arrays. Then data is associated with these sets, such as node coordinates, temperature, pressure etc. The mesh solution is then expressed as the conceptually parallel application of computational kernels on the data associated with each element of a particular set (nodes, edges, cells). These kernels are usually floating point- intensive operations and update the datasets. The procedure is repeated through multiple iterations as desired until a steady-state solution is reached. A more detailed discussion of the unstructured mesh is presented in the Background section of this report.

1.3 FPGAs, streaming and acceleration

In this project we explore the acceleration possibilities of problems in the described domain by using Field Programmable Gate Arrays, or FPGAs. FPGAs are integrated circuits that can be reconfigured on the fly to implement in hardware any logic design. Thanks to this property they provide the development flexibility of software with the benefits of an explicit custom hardware datapath. At a high level, FPGAs can be viewed as a two-dimensional grid of logic elements that can be interconnected in any desirable way.

The FPGA acceleration approach we look at is the streaming model of comutation. In a streaming approach we create a dataflow graph out of simple computational nodes that perform a specific operation on pieces of

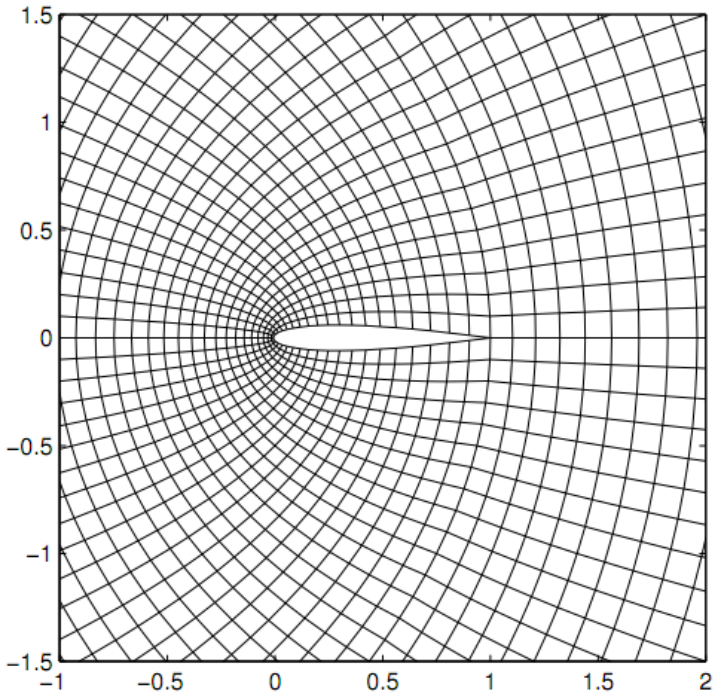

Figure 1.1: Visualisation of a reduced version of the Airfoil mesh

data pushed in and out of them. Connecting these nodes together creates a pipeline through which one can stream an array of data and get one output per cycle thus achieving high throughput. A simple dataflow graph can be seen in figure 1.2. FPGAs are usually programmed using a low level hardware description language like VHDL or Verilog, however many tools have been designed that allow a developer to specify high-level designs. We use MaxCompiler, a compiler that lets us specify the computational graph through a high-level Java API, so we focus on the functional aspects of our design and the tool generates a hardware implementation of it. We use this approach to implement a datapath the kernel described in Airfoil and we then look at approaches to utilise the streaming bandwidth. The FPGAs we consider have a large DRAM storage area attached (¿24GB) to them that can be used to store the mesh and utilising the bandwidth of that DRAM fully is key to achieving maximum performance.

During the course of our work it emerges that in order to stream data to and from the accelerator continuously, we need to enforce some spatial locality in the mesh data, thus requiring us to reorder the data and or-

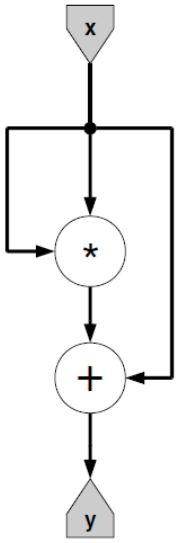

Figure 1.2: A simple dataflow graph that implements the function $y(x) = x^2 + x$.

ganise it into partitions that will be stored in the kernel internally and will need to exchange data with neighbouring partitions through a halo exchange mechanism. This opens a whole new space of decisions that we must make pertaining to the storage layout and streaming responsiblities of the DRAM and the host machine. We present the mesh partitioning schemes that are used to maximise DRAM bandwidth utilisation and maximise pipelining.

We present a performance model that will be used to describe the theoretical performance increase of the system in terms of various parameters like DRAM utilisation, clock frequency etc. Finally we evaluate the performance of our implementation of Airfoil against existing GPGPU and multi-processors cluster implementations.

1.4 Contributions

- We present a methodology for accelerating unstructured mesh computations using deeply pipelined streaming FPGA designs.
- We investigate memory layout issues that arise from efforts to maximise the spatial locality of the mesh.

- We provide a hardware accelerated version of the Airfoil program using the methodologies described in this report.
- We provide a predictive performance model that is used to justify our design decisions and provide a formal expression of the potential speedup.
- We investigate the potential for generalisation of the problem and the acceleration of more complex industry-grade unstructured mesh simulations.

Chapter 2

Background

This section provides more detail on the sample application. An overview of the Maxeler toolchain is given, which is used to implement the streaming solution we develop. The streaming model of computation is presented in the context of MaxCompiler by walking through steps to build a simple MaxCompiler application. Previous work in this area is presented and summarised in order to provide a context for the contributions of our work.

2.1 Unstructured meshes and their representation

The spatial domain of the problem can be discretised into either a structured or an unstructured mesh. A structured mesh has the advantage of having a highly regular structure and thus a highly predictable access pattern. If, however, one needs a more detailed solution around a particular area, the mesh would would have to be fine-grained across the whole domain, thus increasing the number of cells, nodes and edges by a large factor even in areas that are not of such great interest. This is where unstructured meshes come in. They explicitly describe the connectivity between the elements and can thus be refined and coarsened around particular areas of interest. This provides much greater flexibility at the expense of losing the regularity of the mesh, forcing us to store the connectivity information that defines its topology. It is useful to have an intimate understanding of the representation of unstructured meshes in order to understand the techniques discussed further on. A graphical example is shown in figure 2.1

In our sample application the mesh distinguishes three main elements: nodes, cells and edges. We have to represent the connectivity information between them. This is done through *indirection maps* that store, for exam-

ple, the nodes that an edge connects or the nodes that a cell contains. In the application we explore the cells always have four nodes and the edges always connect two nodes and have two cells adjacent. In the more general case of variable-dimension cells (quadrilaterals, triangles, hexagons all mixed together) we would need an additional array storing the indices into the indirection maps and the sizes of the elements. But we do not consider such meshes here.

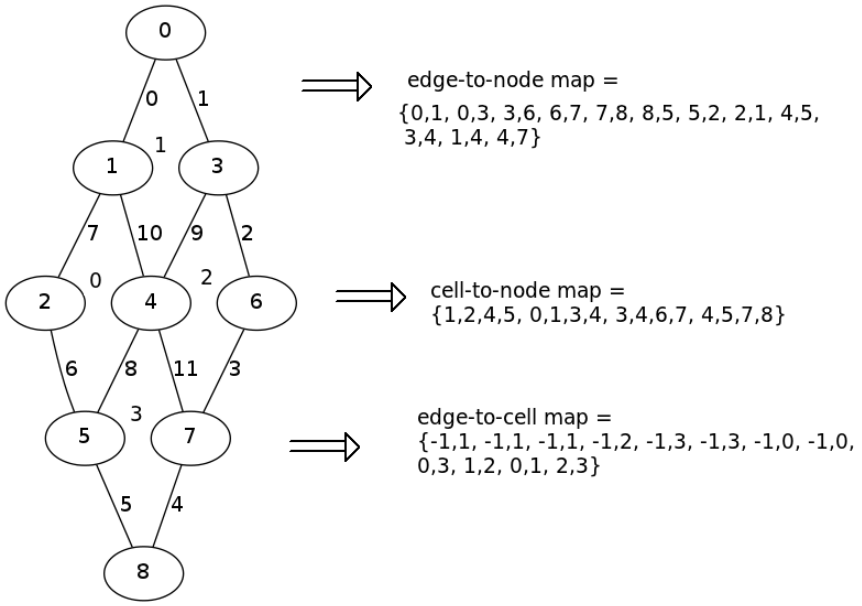

Figure 2.1: An example mesh and its representation using indirection arrays. The cell numbers are shown inside the quadrilaterals formed by the nodes (circles) and edges (edges connecting the nodes). Together with the indirection map, we also store an integer $dim \in \mathbb{N}$ which specifies the dimension of the mapping. Thus, the data associated with element i are stored in the range $[i * dim, \ldots, i * (dim + 1) - 1]$ of the relevant indirection map (in the example: the nodes associated with edge 3 are stored at indices $3 * 2 = 6$ and $3 * 2 + 1 = 7$). Note: in the edge-to-cell map -1 represents a boundary cell that may be handled in a special way by a computational kernel.

The above method deals with the connectivity information amongst the different elements of the mesh. The data on which we perform the actual arithmetic calculations is stored in arrays indexed by element number. Such an approach is presented in figure 2.2.

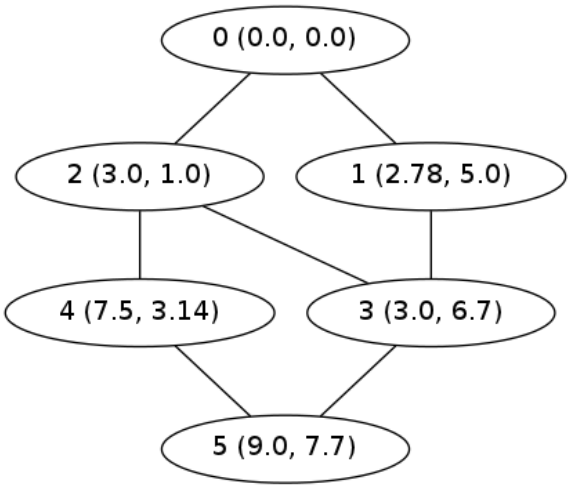

Figure 2.2: An example mesh with coordinate data associated with each node ((x, y) from *node_id* (x, y)). The coordinate data will be represented as an array of floating point numbers $x = \{0.0, 0.0, 2.78, 5.0, 3.0, 1.0, 3.0, 6.7, 7.5, 3.14, 9.0, 7.7\}$. Again we also record the dimension of the data (in this case $dim = 2$) in order to access the data set associated with each element. In this example, the coordinate data for node 4 is stored at indices $4 * 2 = 8$ and $4 * 2 + 1 = 9$ of the array x.

2.2 Airfoil

Airfoil was written as a representative of the class of problems we are interested in. It was initially designed as a non-trivial example of the issues tackled by the OP2 framework. Although we are not directly dealing with OP2 in this project, an overview of Airfoil within this context is provided by MB Giles et al [1] because it discusses the acceleration issues arising from the memory access pattern.

The computational work in Airfoil is performed by 5 loops that work one after the other and operate on the nodes, cells and edges of the mesh. They work by applying a kernel on the data item referenced by the node, edge or cell. Conceptually, the application of a kernel to a data item is independent of the application to any other item in the same set, and can therefore be executed in parallel. The complexity comes from reduce operations, where some edges or cells update the same data item (associated with the same node). In these cases care must be taken to ensure the correct update of the data. For parallel architectures such as GPUs and multi-processor clusters

this issue can be resolved by enforcing an atomic commit scheme or by colouring the mesh partitions, so that no two partitions update the same data item simultaneously [1].

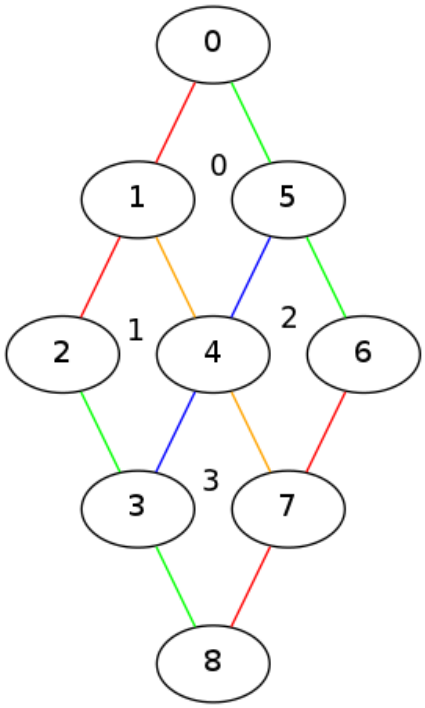

Figure 2.3: An example mesh, showing data dependencies between edges that affect cell data.

Consider figure 2.3. Take for example edges $\alpha = (1, 4)$ and $\beta = (4, 5)$. Say there is a data item x associated with every cell and the processing of an edge increments the data items associated with it's two cells. α and β cannot execute in parallel because they are both associated with cell 0 and can therefore end up using out of date copies of the data associated with cell 0 by the following sequence of events: α reads initial x_0, β reads x_0, α computes $x_\alpha = x_0 + 1$, β computes $x_\beta = x_0 + 1$, α writes back x_α, β writes back x_β and the final value of x turns out to be $x_\beta = x_0 + 1$ instead of the desired $x_0 + 2$. Some implementations work around this issue by colouring the edges, such that no two edges of the same colour share a cell and can therefore be processed in parallel. Figure 2.3 shows such a colouring. Another option would be to introduce atomic operations and/or locking, but

that would approach severely limits parallelisation opportunities.

2.3 Maxeler toolchain and streaming model of computation

The toolchain we use for implementing the FPGA accelerator is the one developed and maintained by Maxeler Technologies. It consists of the MAX3 cards that contain a Xilinx Virtex-6 chip [2] and up to 48GB of DDR3 DRAM. These cards can be programmed through MaxCompiler[3], which provides a Java-compatible object-oriented API to specify the dataflow graph. MaxCompiler will then schedule the graph, i.e. it will insert buffers that will introduce the appropriate delays in the design that will ensure the correct values will reach the appropriate stages in the pipeline at the correct clock cycle.

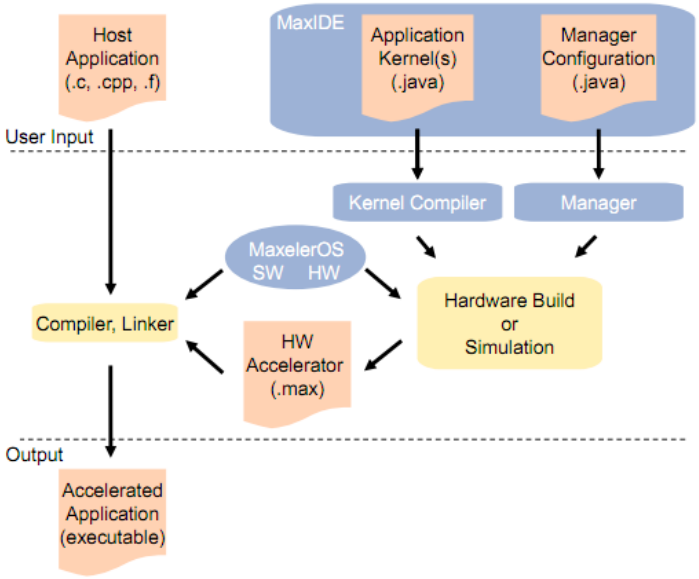

Figure 2.4: A diagram of the Maxeler toolchain. The data-flow graphs of the computational kernels are defined using a Java API. A manager connects multiple kernels together and handles the streaming to and from the kernels of data. These are combined by MaxCompiler and compiled into a .max file that can then be linked to a host C/C++ or Fortran application using standard tools (gcc, ld etc).

It will then produce a hardware design in VHDL that will then be further be compiled down to a binary bitstream that configures the FPGA by the Xilinx proprietary tools. The bitstream is then included in what is termed a *maxfile* that contains various other meta-data about the design such as I/O stream names, named memory and register names, various runtime parameters etc. The maxfile can be linked against a normal C/C++ application using standard tools (gcc, ld etc). The interaction with the FPGA is performed by a low-level runtime: MaxCompilerRT and a driver layer: MaxelerOS. A diagram of the toolchain is shown in figure 2.4 [3].

Computational kernels in MaxCompiler have input streams that are pushed through a pipelined dataflow graph and some of them are output from the kernel. Programmatically, a hardware stream is seen as analogous to a variable in conventional programming languages. It's value potentially changes each cycle.

We present a MaxCompiler design that computes a running 3-point average of a stream of floating point values (32 bits) in Listing 1.

```
1  pulic class MovingAverageKernel extends Kernel {
2
3    public MovingAverageKernel(KernelParameters parameters) {
4      super(parameters);
5      HWType flt = hwFloat(8,24);
6      HWVar x = io.input("x", flt ) ;
7      HWVar x_prev = stream.offset(x, -1);
8      HWVar x_next = stream.offset(x, +1);
9      HWVar cnt = control.count.simpleCounter(32, N);
10     HWVar sel_nl = cnt > 0;
11     HWVar sel_nh = cnt < (N-1);
12     HWVar sel_m = sel_nl & sel_nh;
13     HWVar prev = sel_nl ? x_prev : 0;
14     HWVar next = sel_nh ? x_next : 0;
15     HWVar divisor = sel_m ? 3.0 : 2.0;
16     HWVar y = (prev+x+next)/divisor;
17     io.output("y" , y,  flt);
18   }
19 }
```

Listing 1: A MaxCompiler definition of a kernel that computes a moving 3-point average with boundary conditions. Note that the arithmetic operators as well as the ternary if operator have been overloaded for HWVar objects that represent the value of a hardware stream.

MaxCompiler code is written in a Java-like language called MaxJ that provides overloaded operators such as $+, -, *, /$ and ? : . The example in Listing 1 creates a computational kernel that computes a stream of running 3-point averages,named y, from a stream of input values x. The HWVar class is the main representation of the value of a hardware stream at any clock cycle. HWVars always have a HWType that expresses the type of the stream (i.e. an integer, a floating point number, a 1-bit boolean value etc). The $stream.offset(x, -1)$ and $stream.offset(x, +1)$ expressions on lines 7 and 8 extract HWVars for the values of the stream on cycle in the past and one cycle in the future (note that this is internally done by creating implicit buffers, or FIFOs, and scheduling the pipeling accordingly). The ternary if operator ? : creates multiplexers in hardware that express choice. A Java API is provided that contains various useful design elements, such as counters (HWVars that increment their values in many configurable ways every cycle) that can be accessed through the control.count field.

The resulting control flow graph can be seen in figure 2.5

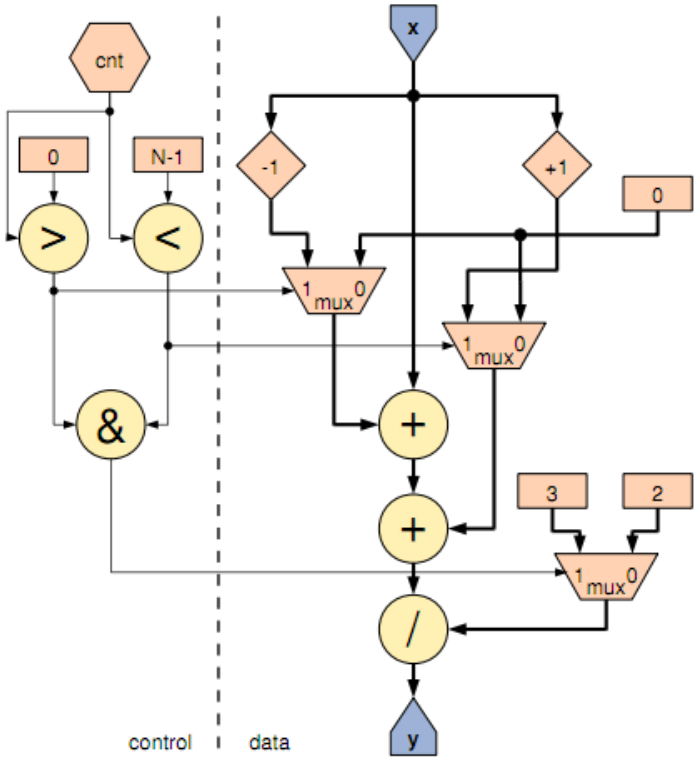

Figure 2.5: The dataflow graph resulting from the code in Listing 1

Kernel designs form part of a MaxCompiler design. The user also specifies a manager that describes the streaming connections between the kernels. A manager can be used to configure a design to stream data to and from the host through PCIe or from the DRAM that is attached to the FPGA. In the manager design, the user will instantiate the kernels and connect them up. Thus for the example in Listing 1 the manager might look like the one in Listing 2.

```
pulic class MovingAvgManager extends CustomManager {

  public MovingAvgManager(MAXBoardModel board_model,
                          boolean is_simulation, String name) {
    super(is_simulation, board_model, name);
    KernelBlock k
      = addKernel(
          new MovingAverageKernel(makeKernelParameters("MovingAverageKernel"))
                );

    Stream x = addStreamFromHost("x");
    k.getInput("x") <== x;

    Stream y = addStreamToHost("y");
    y <== k.getOutput("y");
  }
}
```

Listing 2: Manager specification for a MovingAverageKernel that streams the input data "x" from the host and streams the output data "y" to the host. The <== operator means connect the right hand side stream to the left hand side stream. The above code instantiates the MovingAverageKernel, creates a stream called "x" from the host and connects it to the input stream "x" in the kernel. Then it creates a stream to the host called "y" and connects to it the output stream "y" from the kernel.

After we have specified a manger, we can build the design in order to create the .max file using the following lines of code:

```
public class MovingAvgHWBuilder {
  public static void main(String argv[]) {

    MovingAvgManager m
      = new MovingAvgManager(MAX3BoardModel.MAX3242A,
                             false,
                             "MovingAverage");
    m.build() ;
  }
}
```

This builds our design for a MAX3 card (containing a Xilinx Virtex6 FPGA)

using the "MovingAverage" name for the design.

Now that we have a .max file, we can interact with the FPGA from the host code by using the MaxCompilerRT API, an example of which is shown in Listing 3.

```
1  #include<stdlib.h>
2  #include<stdint.h>
3  #include<MaxCompilerRT.h>
4  #define DATA_SIZE 1024
5
6  int main(int argc, char* argv[]) {
7          char* device_name = "/dev/maxeler0";
8          max_maxfile_t* maxfile;
9          max_device_handle_t* device;
10         float *data_in, *data_out;
11
12         maxfile = max_maxfile_init_MovingAverage();
13         device = max_open_device(maxfile, device_name);
14
15         data_in = (float*)malloc(DATA_SIZE * sizeof(float));
16         data_out = (float*)malloc(DATA_SIZE * sizeof(float));
17
18         for (int i = 0; i < DATA_SIZE; ++i) {
19                 data_in[i] = i;
20         }
21
22         max_run(device,
23                 max_input("x", data_in, DATA_SIZE * sizeof(float)),
24                 max_output("y", data_out, DATA_SIZE * sizeof(float)),
25                 max_runfor("MovingAverageKernel", DATA_SIZE),
26                 max_end());
27
28
29         for (int i = 0; i < DATA_SIZE; ++i) {
30                 printf("data_out@%d = %f\n", i, data_out[i]);
31         }
32
33         max_close_device(device);
34         max_destroy(maxfile);
35         return 0;
36
37 }
```

Listing 3: A sample host code using the MaxCompilerRT API for the C language. In order to use the FPGA we must initialise the maxfile as in line 14 and open the device (line 15). The actual streaming to and from the FPGA is done using the max_run vararg function (line 22) where the arrays corresponding to the input data and the allocated space for the output data are specified. The MaxCompilerRT runtime and the MaxelerOS drivers handle the low-level details of PCIe streaming and interrupts.

2.4 Previous work

There have been some attempts at augumenting unstructured mesh computations using reconfigurable coprocessors, although these attempts are not very common because of the irregular memory access patterns and potentially complicated data dependenciesthat are considered to be an undesirable characteristic for FPGA acceleration, in particular the streaming model of computation where we want to push a data item to the computational kernel every cycle and get one result per cycle. M.T. Jones and K.Ramachandran [5] formulate the unstructured mesh computation as a sparse matrix problem, $Ax = y$ where A is a large sparse matrix representing the mesh and x is the vector that is being computer/approximated. Their approach uses the conjugate gradient method to iteratively refine the approximation of the x vector. This involves, most importantly, a multiplication of the sparse matrix A with the vector x which forms the bulk of the computation and a subsequent refinement of the mesh and reconstruction of the spares matrix. Our approach differs from theirs in that we assume a static mesh that is not refined. Also, we iterate for a constant number of times, with no convergence criteria, we only care about the values of the data sets that are declared over the nodes, edges and cells of the mesh. We do not construct but instead apply a number of kernels on each element of the mesh in turn.

Morishita et al. [6] also examine the acceleration of CFD applications and in particular the use of on-chip block RAM resources to buffer the data in order to keep the arithmetic pipeline as full as possible. This is more similar approach. However, their approach applies a constant stencil to a grid in 3D and tries to cache points in the grid that will be accessed in the next iteration, thus elimintating redundant accesses to the external memory. This caching/buffering is made possible by the fact that the stencil is of constant shape and thus the memory accesses can be predicted. In our application we have a 2D mesh that does not exhibit this property.

Sanchez-Roman et al. [7] present the acceleration of an airfoil-like unstructured mesh computation using FPGAs. Their solution uses two FPGAs on a single chip that perform different calculations. They mention the need to partition larger meshes but they do not discuss techniques for partitioning or the issues arising from data dependencies across partitions. Also they do not present any formal analysis of the performance of their solution that could be used to predict speedup.

Chapter 3

Details and implementation

In this section we present our design, implementation and evaluation plan and provide the details of each stage.

3.1 Plan

We start by proposing various architectures for solving the problem and we propose a formal model for each one that allows us to predict the performance of the architecture. We then implement our scheme in order to provide real-world results and assess the feasibility of the implementation. This model will also allow us to pick the optimal values of the parameters of the application, thus maximizing performance and saving us the effort of using a trial and error approach to fine tune them. After that we proceed with the implementation of our chosen architecture.

During our work we realize that we have to partition the mesh into chunks that we can fit into the on-chip memory (called BRAM or block RAM) for processing. This requires us to think about and deal with data that overlap partitions (for example edges that begin in one partition and end in another). The set of shared data is known as the 'halo' of the partition and various halo exchange schemes exist. We use the ghost cell exchange mechanism, presented in [4].

After we have decided on the various parameters of the design (partition size, number of arithmetic pipelines, streaming responsibilities etc) we implement our design using MaxCompiler to produce a maxfile that we link to the Airfoil executable that we have modified to partition and layout the data in the decided way.

We can then compare our implementation against the theoretical model

developed earlier and track down and explain any and all discrepancies. Then we can compare our implementation against existing implementations that use Nvidia's OpenCL CUDA implementation.

Bibliography

[1] MB Giles, GR Mudalige, Z Sharif, G Markall, PHJ Kelly,
Performance Analysis of the OP2 Framework on Many-core Architectures.
ACM SIGMETRICS Performance Evaluation Review, 38(4):9-15, March 2011

[2] Xilinx Inc.
Virtex-6 Family Overview
http://www.xilinx.com/support/documentation/virtex-6.htm

[3] Maxeler Technologies
MaxCompiler White Paper
http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxCompiler.pdf

[4] F. B. Kjolstad, M Snir
Ghost Cell Pattern
ParaPLoP '10 Proceedings of the 2010 Workshop on Parallel Programming Patterns

[5] M. T. Jones, K. Ramachandran
Unstructured mesh computations on CCMs
Advances in Engineering Software - Special issue on large-scale analysis, design and intelligent synthesis environments Volume 31 Issue 8-9, Aug-Sept. 2000

[6] H. Morishita, Y. Osana, N. Fujita, H. Amano
Exploiting memory hierarchy for a Computational Fluid Dynamics accelerator on FPGAs
ICECE Technology, 2008. FPT 2008. pp 193 - 200

[7] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, J.; Palacios, F.;

High-Level Languages and Floating-Point Arithmetic for FPGABased CFD Simulations
Design & Test of Computers, IEEE, 2011, Volume: 28 Issue:4, pp 28 - 37

[8] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, A.;
An Euler Solver Accelerator in FPGA for computational fluid dynamics applications
Proceedings of the 2011 VII Southern Conference on Programmable Logic Crdoba, Argentina April 13 15, 2011

[9] Durbano, J.P.; Ortiz, F.E.;
FPGA-based acceleration of the 3D finite-difference time-domain method
12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004.

Chapter 4

Appendix